An Optimistic Mandatory Access Control Model for Distributed Collaborative Editors

Abdessamad Imine, Asma Cherif, Michaël Rusinowitch

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Optimistic Mandatory Access Control Model for Distributed Collaborative Editors

Abdessamad Imine and Asma Cherif and Michaël Rusinowitch

N° 6939

February 2009

Thème SYM

ISSN 0249-6399

ISSN 0249-6399 ISRN INRIA/RR--6939--FR+ENG

An Optimistic Mandatory Access Control Model for Distributed Collaborative Editors

Abdessamad Imine* and Asma Cherif† and Michaël Rusinowitch‡

Thème SYM — Systèmes symboliques
Projet CASSIS

Rapport de recherche n° 6939 — February 2009 — 20 pages

Abstract: Distributed Collaborative Editors (DCE) provide computer support for modifying simultaneously shared documents, such as articles, wiki pages and programming source code, by dispersed users. Controlling access in such systems is still a challenging problem, as they need dynamic access changes and low latency access to shared documents. In this paper, we propose a Mandatory Access Control (MAC) based on replicating the shared document and its authorization policy at the local memory of each user. To deal with latency and dynamic access changes, we use an optimistic access control technique where enforcement of authorizations is retroactive. We show that naive coordination between updates of both copies can create security hole on the shared document by permitting illegal modification, or rejecting legal modification. Finally, we present a novel framework for managing authorizations in collaborative editing work which may be deployed easily on P2P networks.

Key-words: Access Control, Optimistic Replication, Distributed Collaborative Editors.

* INRIA Nancy Grand Est & Univ. Nancy 2, UMR 7503 (imine@loria.fr).
† INRIA Nancy Grand Est & Univ. Nancy 2, UMR 7503 (cheriasm@loria.fr).
‡ INRIA Nancy Grand Est, UMR 7503 (rusi@loria.fr).

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Modèle Optimiste de Contrôle d'Accès Obligatoire pour les Editeurs Collaboratifs

Résumé : Les éditeurs collaboratifs fournissent un support logiciel pour la modification simultanée des documents partagés, comme des articles, des pages wiki et du code source des programmes, par des utilisateurs dispersés géographiquement. Le contrôle d'accès dans de tels systèmes demeure toujours un challenge difficile, car ils nécessitent des accès dynamiques ainsi qu'une faible latence pour accéder aux documents partagés. Dans ce rapport, nous proposons un contrôle d'accès obligatoire qui se base sur la réplication du document partagé ainsi que sa politique d'accès. Pour traiter des problèmes de la latence et les accès dynamiques, nous utilisons une technique de contrôle d'accès optimiste où l'exécution des autorisations est rétroactive. Nous montrons qu'une coordination naïve entre les mises à jour des deux copies peut causer des failles de sécurité en permettant des modifications illégales ou en rejetant des modifications légales. Enfin, nous présentons un nouvel environnement pour la gestion des autorisations dans des éditeurs collaboratifs qui peut être facilement déployé sur des réseaux P2P.

Mots-clés : Contrôle d'accès, Réplication Optimiste, Editeurs Collaboratifs.

Contents

1 Introduction **4**

2 Related Works **5**

3 Our Collaboration Model **5**
3.1 Shared Data Object 5
3.2 Shared Policy Object 6
3.3 Collaboration Protocol 7

4 Consistency and Security Issues **7**
4.1 Out-of-order Execution of Cooperative Operations 7
4.2 Out-of-order Execution of Cooperative and Administrative Operations 8

5 Concurrency Control Algorithm **11**
5.1 Cooperative and Administrative Requests 12
5.2 Control Procedure 12
5.3 Illustrative Example 15

6 Implementation and Evaluation **17**

7 Conclusion **18**

1 Introduction

Distributed Collaborative Editors (DCE) belong to a particular class of distributed systems that enables several and dispersed users to form a group for editing documents (*e.g.* Google Docs). To ensure data availability, the shared documents are replicated on the site of each participating user. Each user modifies locally his copy and then sends this update to other users.

DCE are distributed systems that have to consider human interactions. So, they are characterised by the following requirements: (i) *High local responsiveness*: the system has to be as responsive as its single-user editors [3, 15, 16]; (ii) *High concurrency*: the users must be able to concurrently and freely modify any part of the shared document at any time [3, 15]; (iii) *Consistency*: the users must eventually see a converged view of all copies [3, 15] in order to support WYSIWIS (What You See Is What I See) principle; (iv) *Decentralized coordination*: all concurrent updates must be synchronized in decentralized fashion in order to avoid a single point of failure; (v) *Scalability*: a group must be dynamic in the sense that users may join or leave the group at any time.

Motivations. One of the most challenging problem in DCE is balancing the computing goals of collaboration and access control to shared information [17]. Indeed interaction in collaborative editors is aimed at making shared document available to all who need it, whereas access control seeks to ensure this availability only to users with proper authorization. Moreover, the requirements of DCE include high responsiveness of local updates. However, when adding an access control layer, high responsiveness is lost because every update must be granted by some authorization coming from a distant user (as a central server). The major problem of latency in access control-based collaborative editors is due to using one shared data-structure containing access rights that is stored on a central server. So controlling access consists in locking this data-structure and verifying whether this access is valid. Furthermore, unlike traditional single-user models, collaborative applications have to allow for dynamic change of access rights, as users can join and leave the group in an ad-hoc manner.

Contributions. To overcome the latency problem, we propose to replicate the access data-structure on every site. Thus, a user will own two copies: the shared document and the access data-structure. It is clear that this replication enable users to gain performance since when they want to manipulate (read or update) the shared document, this manipulation will be granted or denied by controlling only the local copy of the access data-structure. As DCE have to allow for dynamic change of access rights, it is possible to achieve this goal when duplicating access rights. To do that, we propose a Mandatory Access Control model (MAC) [11], in the sense that only one user, called *administrator*, can modify the shared access data-structure. Thus, updates locally generated by the administrator are then broadcast to other users. We choose dynamic access changes initiated by one user in order to avoid the occurrence and the resolution of conflict changes.

The shared document's updates and the access data-structure's updates are applied in different orders at different user sites. The absence of safe coordination between these different updates may cause security holes (*i.e.* permitting illegal updates or rejecting legal updates on the shared document). Inspired by the *optimistic security* concept introduced in [8], we propose an optimistic approach that tolerates momentary violation of access rights but then ensures the copies to be restored in valid states with respect to the stabilized access control policy.

Outline of the paper. This paper is organized as follows: Section II discusses related work. Section III presents the ingredients of our collaboration model. In Section IV, we investigate the issues raised by replicating the shared document and its access data-structure. Section V presents our concurrency control algorithm for managing MAC-based collaborative editing sessions. Section VI describes our prototype and its evaluation on experiments. Section VII summarizes our contributions and sketch future works.

2 Related Works

A survey on access control for collaborative systems can be found in [17]. We only recall some representative approaches and their shortcomings. A collaborative environment has to manage the frequent changing of access rights by users. Access Control Lists (ACL) and Capability Lists (CL) cannot support very well dynamic change of permissions. Hence, the administrator of collaborative environments often sets stricter permissions, as multiple users with varying levels of privileges will try to access shared resources [13]. Role Based Access Control (RBAC) [12] overcomes some problems with dynamic change of rights. RBAC has the notion of a session which is a per-user abstraction [5]. However, the "session" concept also prevents a dynamic reassignment of roles since the user roles cannot be changed within a single session. Users have to authenticate again to obtain new roles. Spatial Access Control (SAC) has been proposed to solve this problem of role migration within a session [2]. Instead of splitting users into groups as in RBAC, SAC divides the collaborative environment into abstract spaces. However, SAC implementation needs prior knowledge of the practice used in some collaborative system, in order to produce a set of rules that are generic enough to match most of the daily access patterns. Every access needs to check the underlying access data-structures; this requires locking data-structures and reduces collaborative work performance.

The majority of works on replicating authorization policies appears in database area [10, 1, 18]. For maintaining authorization consistency, these works generally rely on concurrency control techniques that are suitable for database systems. As outlined in [3], these techniques are inappropriate for DCE. Nevertheless, [10] is related to our work as it employs an optimistic approach. Indeed, changes in authorizations can arrive in different order at different sites. Unlike our MAC approach, conflict authorizations may appear as updates are initiated by several sites.

3 Our Collaboration Model

We give here the ingredients of our model.

3.1 Shared Data Object

It is known that collaborative editors manipulate share objects that admit a linear structure [3, 14, 16]. This structure can be modelled by the *list* abstract data type. The type of the list elements is a parameter that can be instantiated by each needed type. For instance, an element may be regarded as a character, a paragraph, a page, an XML node, etc. In [16], it has been shown that this linear

structure can be easily extended to a range of multimedia documents, such as MicroSoft Word© and PowerPoint© documents.

Definition 3.1 ***[Cooperative Operations].*** *The shared document state can be altered by the following set of* cooperative operations*: (i) $Ins(p,e)$ where p is the insertion position, e the element to be added at position p; (ii) $Del(p,e)$ which deletes the element e at position p; (iii) $Up(p,e,e')$ which replaces the element e at position p by the new element e'.* □

It is clear that combinations of these operations enable us to define more complex ones, such as cut/copy and paste, that are intensively used in professional text editors.

3.2 Shared Policy Object

We consider an access control model based on *authorization policies*. An authorization policy specifies the operations a user can execute on a shared document. Three sets are used for specifying authorization policies, namely:

1. S is the set of *subjects*. A subject can be a user or a group of users.
2. O is the set of *objects*. An object can be the whole shared document, an element or a group of elements of this shared document.
3. R is the set of *access rights*. Each right is associated with an operation that user can perform on shared document. Thus, we consider the right of reading an element (rR), inserting an element (iR), deleting an element (rD) and updating an element (rU).

Definition 3.2 ***[Policy].*** *A* policy *is a function that maps a set of subjects and a set of objects to a set of signed rights. We denote this function by $P : \mathcal{P}(S) \times \mathcal{P}(O) \rightarrow \mathcal{P}(R) \times \{+,-\}$, where $\mathcal{P}(S)$, $\mathcal{P}(O)$ and $\mathcal{P}(R)$ are the power sets of subjects, objects and rights respectively. The sign "$+$" represents a right* attribution *and the sign "$-$" represents a right* revocation. □

We represent a policy P as an indexed list of authorizations. Each authorization P_i is a quadruple $\langle S_i, O_i, R_i, \omega_i \rangle$ where $S_i \subseteq S$, $O_i \subseteq O$, $R_i \subseteq R$ and $\omega_i \in \{-,+\}$. An authorization is said *positive* (resp. *negative*) when $\omega = +$ (resp. $\omega = -$). Negative authorizations are just used to accelerate the checking process. We use a first-match semantics: when an operation o is generated, the system checks o against its authorizations one by one, starting from the first authorization and stopping when it reaches the first authorization l that matches o. If no matching authorizations are found, o is rejected.

Definition 3.3 ***[Administrative Operations].*** *The state of a policy is represented by a triple $\langle P,S,O \rangle$ where P is the list of authorizations. The administrator can alter the state policy by the following set of* administrative operations*: (i) $AddUser/DelUser$ to add/remove a user in S; (ii) $AddObj/DelObj$ to add/remove an object in O; (iii) $AddAuth(p,l)/DelAuth(p,l)$ to add/remove authorization l at position p. An administrative operation r is called* restrictive *iff $r = AddAuth(p,l)$ and l is negative or $r = DelAuth(p,l)$.* □

3.3 Collaboration Protocol

In our collaboration protocol, we consider that a user maintains two copies: the shared document and its access policy object. Each group consists of one administrator and several users. Only administrator can specify authorizations in the policy object. It can also modify directly the shared documents. As for users, they only modify the shared document with respect to the local policy object. Our collaboration protocol proceeds as follows:

1. When a user manipulates the local copy of the shared document by generating a cooperative operation, this operation will be granted or denied by only checking the local copy of the policy object.

2. Once granted and executed, the local operations are then broadcast to the other users. A user has to check whether or not the remote operations are authorized by its local policy object before executing them.

3. When an administrator modifies its local policy object by adding or removing authorizations, he sends these modifications to the other users in order to update their local copies. Note that the administrator site does not coordinate concurrent cooperative operations.

We assume that messages are sent via secure and reliable communication network, and users are identified and authenticated by the administrator in order to associate correctly access to these users.

4 Consistency and Security Issues

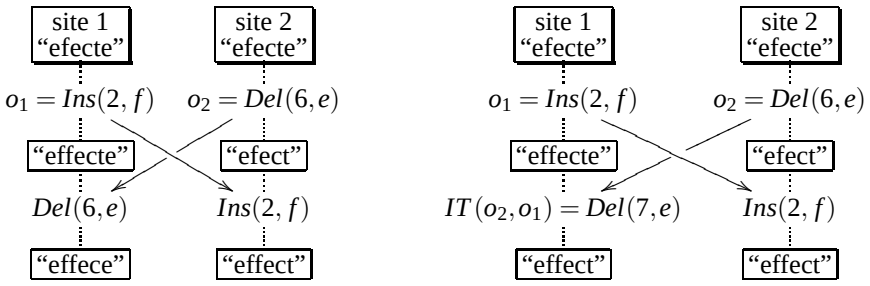
The replication of the shared document and the policy object is twofold beneficial: firstly it ensures the availability of the shared document, and secondly it allows for flexibility in access rights checking. However, this replication may create violation of access rights which may fail to meet one of the most important requirements of DEC, the consistency of the shared document's copies. Indeed, the cooperative and administrative operations are performed in different orders on different copies of the shared document and the policy object.

In the following, we investigate the issues raised by the use of the collaboration protocol described in Section 3.3 and we informally present our solutions to address these issues.

4.1 Out-of-order Execution of Cooperative Operations

What happens if cooperative operations arrive in arbitrary orders even with stable policy object? Consider the scenario in Figure 1.(a) where two users work on a shared document represented by a sequence of characters and they have the same policy object (they are authorized to insert and delete characters). These characters are addressed from 1 to the end of the document. Initially, both copies hold the string "efecte". User 1 executes operation $o_1 = Ins(2, f)$ to insert the character 'f' at position 2. Concurrently, user 2 performs $o_2 = Del(6, e)$ to delete the character 'e' at position 6. When o_1 is received and executed on site 2, it produces the expected string "effect". But, at site 1, o_2 does not take into account that op_1 has been executed before it and it produces the string "effece".

The result at site 1 is different from the result of site 2 and it apparently violates the intention of o_2 since the last character 'e', which was intended to be deleted, is still present in the final string.



(a) Incorrect integration. (a) Correct integration.

Figure 1: Serialization of concurrent cooperative operations

To maintain consistency of the shared document, even though the policy object remains unchanged, we use the Operational Transformation (OT) approach which has been proposed in [3]. In general, it consists of application-dependent transformation algorithm, called IT, such that for every possible pair of concurrent operations, the application programmer has to specify how to integrate these operations regardless of reception order. In Figure 1.(b), we illustrate the effect of IT on the previous example. At site 1, o_2 needs to be transformed in order to include the effects of o_1: $o_2' = IT((Del(6,e), Ins(2,f)) = Del(7,e)$. The deletion position of o_2 is incremented because o_1 has inserted a character at position 1, which is before the character deleted by o_2. It should be noted that OT enables us to ensure the consistency for *any number* of concurrent operations which can be executed in *arbitrary order* [9, 7] (*i.e.* no global order is necessary).

For managing collaborative editing work in a decentralized and scalable fashion, we reuse an OT-based framework that is not presented here due to space limit. For more details see *e.g.* [4]. Our objective here is to develop on the top of this framework a security layer for controlling access to the shared documents.

4.2 Out-of-order Execution of Cooperative and Administrative Operations

Performing cooperative and administrative operations in different orders at every user site may inevitably lead to security holes. To underline these issues we will present in the following three scenarios.

First scenario: Consider a group composed of an administrator *adm* and two standard users s_1 and s_2. Initially, the three sites have the same shared document "abc" and the same policy object where s_1 is authorized to insert characters (see Figure 2). Suppose that *adm* revokes the insertion right of s_1 and sends this administrative operation to s_1 and s_2 so that it is applied on their local policy copies. Concurrently s_1 executes a cooperative operation $Ins(1,x)$ to derive the state "xabc" as it is granted by its local policy. When *adm* receives the s_1's operation, it will be ignored (as it is not granted

by the *adm*'s local policy) and then the final state still remain "abc". As s_2 receives the s_1's insert operation before its revocation, he gets the state "xabc" that will be unchanged even after having executed the revocation operation. We are in presence of data inconsistency (the state of *adm* is different from the state of s_1 and s_2) even though the policy object is same in all sites.

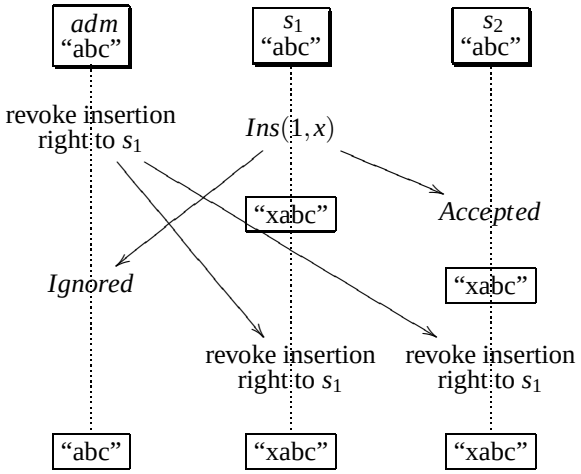


Figure 2: Divergence caused by introducing administrative operations

The new policy object is not uniformly enforced among all sites because of the out-of-order execution of administrative and cooperative operations. Thus, security holes may be created. For instance some sites can accept cooperative operations that are illegal with respect to the new policy (*e.g.* sites s_1 and s_2).

As our objective is to deploy such DCE in a P2P environment, the solution based on enforcing a total order between both operations is discarded as it would require a central server. Achieving this objective raises a critical question: how the enforcement of the new policy is performed with respect to concurrent cooperative operations? It should be noted that this enforcement may be delayed by either the latency of the network or malicious users.

To solve this problem, we apply the principles of optimistic security [8] in such a way that the enforcement of the new policy may be retroactive with respect to concurrent cooperative operations. In this case, only illegal operations are undone. For instance, in Figure 2, $Ins(1,x)$ should be undone in s_1 and s_2 after the execution of the revocation.

Second scenario: Suppose now that we use some technique to detect concurrency relations between administrative and cooperative operations. In the scenario of Figure 3, three users see initially the same document "abc" and they use the same policy object $P =< \{s_2\},\{doc\},\{rD\},+ >$. Firstly, *adm* revokes the deletion right to s_2 by removing an authorization from P (P becomes empty). Concurrently, s_2 performs $Del(1,a)$ to obtain the state "bc". Once the revocation arrives at s_2, it updates the local policy copy and it enforces the new policy by undoing $Del(1,a)$ and restoring the state to "abc".

How to integrate the remote operation $Del(1,a)$ at adm and s_1? Before to execute this operation, if we check it directly against the local policy at adm, it will be rejected (the policy is empty). After a while of receiving and ignoring operation $Del(1,a)$, adm decides to grant once again the deletion right to s_2. At s_1, the execution of both administrative operations leads to $P =< \{s_2\},\{doc\},\{rD\},+ >$. Before to execute $Del(1,a)$, if we check it directly with respect to the local policy of s_1 then it will be granted and its execution will lead to data inconsistency.

This security hole comes from the fact that the generation context of $Del(1,a)$ (the local policy on which it was checked) at s_2 is different from the current execution context at adm and s_1 (due to preceding executions of concurrent administrative operations).

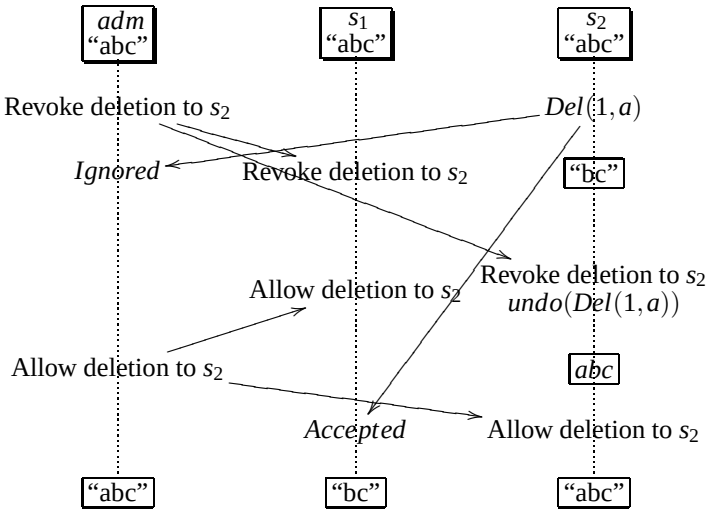


Figure 3: Necessity of admin Log.

Intuitively, our solution consists in capturing the causal relations between cooperative operations and the policy copies on which they are generated. In other words, every local policy copy maintains a monotonically increasing counter that is incremented by every administrative operation performed on this copy. If each granted cooperative operation is associated with the local counter of the policy object at the time of its creation, then we can correctly integrate it in every remote site. However, when the cooperative operation's counter is less than the policy copy's counter of another site then this operation need to be checked with respect to preceding concurrent administrative operations before its execution. Therefore, we propose in our model to store administrative operations in a log at every site in order to validate the remote cooperative operations at appropriate context. For instance, in Figure 2, we can deduce that $Del(1,a)$ will be ignored at s_1 by simply checking it against the first revocation.

Third scenario: Using the above solution, the administrative operations will be totally ordered as only administrator modifies the policy object and we associate to every version of this object a monotonically increasing counter.

Consider the scenario illustrated in Figure 4 where s_1 is initially authorized to insert any character. When *adm* revokes the insertion right to s_1, he has already seen the effect of the s_1's insertion. If s_2 receives the revocation before the insertion, he will ignore this insertion as it is checked against the revocation. It is clear that the insertion may be delayed at s_2 either by the latency of the network or by a malicious user. We observe that there is a causal relation at *adm* between the insertion and the revocation. This causal relation is not respected at s_2 and the out-of-execution of operations creates a security hole as s_2 rejects a legal insertion.

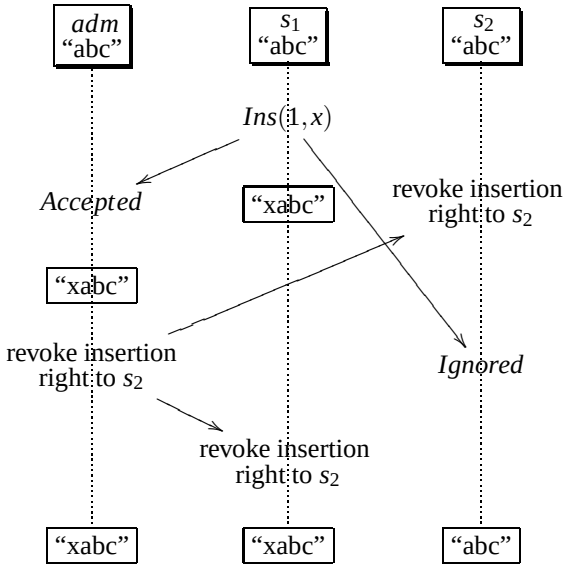


Figure 4: Validation of operations

Before it is received at the administrator site, we consider a cooperative operation as tentative. So, our solution consists of an additional administrative operation that doesn't modify the policy object but increments the local counter. This operation validates each received and accepted cooperative operation at the administrator site. Consequently, every administrative operation is concurrent to all tentative operations. The policy modifications done after the validation of a cooperative operation are executed after this operation in all sites, as administrative operations are totally ordered.

In case of our scenario in Figure 4, the revocation received at s_2 will not be executed until the validation of the insertion is received. This avoids blocking legal operations and data divergence.

5 Concurrency Control Algorithm

Now we formally present the different components of our algorithm. We also give its asymptotic time complexity.

5.1 Cooperative and Administrative Requests

We define a *cooperative request* q as a tuple (c, r, a, o, v, f) where: (i) c is the identity of the collaborator site (or the user) issuing the request. (ii) r is its serial number (note that the concatenation of $q.c$ and $q.r$ is defined as the request identity of q). (iii) a is the identity of the preceding cooperative request[1]. If a is *null* then the request does not depend on any other request. (iv) o is the cooperative operation (see Definition 3.1) to be executed on the shared state. (v) v is the number version of the policy copy on which the operation is granted. (vi) f is the kind of cooperative (tentative, valid or invalid). We consider three kinds of cooperative requests:

1. *tentative* : when an operation is locally accepted, it is stored as a request waiting for validation from the administrator.
2. *valid* : it is generated by a given site and validated by the local policy of the administrator.
3. *invalid* : this means that it is not confirmed by the receiver local policy. It is then stored in the log and flagged in order to memorize its reception.

To detect causal dependency and concurrent relations between cooperative requests, we use a technique proposed in [4] which allows for dynamic groups as it is *independent* of the number of users (unlike to vector timestamp-based technique [3]). This technique builds a dependency tree where each request q has only to store in $q.a$ the request identity whose it directly depends on. For more details, see [4].

We consider an *administrative request* r as the triple $r = (id, o, v)$ where: (i) id is the identity of the administrator; (ii) o is the administrative operation (see Definition 3.3); (iii) v is the last version number of the policy object. As only administrator specifies authorizations in the policy object, the administrative requests are totally ordered. Indeed, each policy copy maintains a monotonically increasing counter that is stored (in the version component v) and incremented by every administrative operation performed on this copy.

As seen in Section 4, it is crucial to correctly deal with the out-of-order execution between cooperative and administrative requests in order to avoid the security holes. Let q and r be cooperative and administrative requests respectively: (i) q depends causally on r iff $q.v > r.v$, *i.e.* q already has seen the effect of r; (ii) if q is tentative then it is concurrent to r, *i.e.* the administrator has not yet seen the effect of q when it generates r.

5.2 Control Procedure

In our approach, a group consists of one administrator site and N user sites (where N is variable in time) starting a collaboration session from the same initial document state D_0. Each site stores all cooperative requests in log H and administrative requests (*AddAuth* and *DelAuth*) in a log L. Our concurrency control procedure is given in Algorithm 1. It should be noted that Algorithm 1 is mainly based on framework proposed in [4]. This framework relies on (i) using OT approach [3] in order to execute cooperative requests in any order; (ii) using a particular class of logs, called *canonical*, where insertion requests are stored before deletion requests in order to ensure data convergence.

[1] According to the dependency relation described in [4].

```
1: Main:
2: INITIALIZATION
3: while not aborted do
4:     if there is an input o then
5:         if o is cooperative then
6:             GENERATE_COOP_REQUEST
7:         else if i = admin then
8:             GENERATE_ADMIN_REQUEST
9:         end if
10:    else
11:        RECEIVE_REQUEST
12:        RECEIVE_COOP_REQUEST
13:        RECEIVE_ADMIN_REQUEST
14:    end if
15: end while

16: INITIALIZATION:
17: D ← D_0 {Actual state of the site}
18: s ← Identification of local site
19: version ← 0 {Initial Version of local site}
20: H ← [] {Cooperative log}
21: L ← [] {Administrative log}
22: F ← [] {Cooperative requests buffer}
23: Q ← [] {Administrative requests buffer}
24: compteurCoopOp ← 0

25: RECEIVE_REQUEST:
26: if there is a cooperative request q from a network then
27:     F ← F + q
28: end if
29: if there is an administrative request r from a network then
30:     Q ← Q + r
31: end if
```

Algorithm 1: Control Concurrency Algorithm at the i-th site

Generation of local cooperative request. In Algorithm 2, when an operation o is locally generated, it is first checked against the local policy object (*i.e.* using boolean function CHECK_LOCAL). If it is granted locally, it is immediately executed on its generation state (*i.e.* $Do(o,D)$ computes the resulting state when executing operation o on state D). Once the request q is formed, it is considered either as valid when the issuer is the administrator or otherwise as tentative. Function COMPUTEBF(q,L) is called to detect inside H whether or not q is causally dependent on precedent cooperative request. Integrating q after H may result in not canonical log. To transform $[H;q]$ in canonical form, we use function CANONIZE. Finally, the request q' (the result of COMPUTEBF) is propagated to all sites in order to be executed on other copies of the shared document. For more details on functions COMPUTEBF and CANONIZE, see [4].

Reception of cooperative request. Each site has the use of queue F to store the remote requests coming from other sites. Request q generated on site i is added to F when it arrives at site j (with $i \neq j$). In Algorithm 3, to preserve the causality dependency with respect to precedent administrative requests and precedent cooperative requests, q is extracted from the queue when it is *causally-ready* (*i.e.* $q.v \leq version$ and the precedent cooperative requests of q have been already integrated on site j). Using function CHECK_REMOTE(q,L), q is checked against the administrative log L to verify whether or not q is granted. If q is received by the administrator then it is validated and a validation

```
1: GENERATE_COOP_REQUEST:
2: if CHECK_LOCAL(o) then
3:     D ← Do(o,D)
4:     compteurCoopOp ← compteurCoopOp + 1
5:     if i = admin then
6:         q ← (s, compteurCoopOp, o, null, Valid, version)
7:     else
8:         q ← (s, compteurCoopOp, o, null, Tentative, version)
9:     end if
10:    q' ←COMPUTEBF(q,H)
11:    H ← CANONIZE(q',H)
12:    broadcast q';
13: end if
```

Algorithm 2: Cooperative request generation at a site s

request is generated in order to broadcast it to other sites. Next, function COMPUTEFF(q,L) is called in order to compute the transformed form q' to be executed on current state D. This function is given in [4]. Finally, the transformed form of q, namely q', is executed on the current state and function CANONIZE is called in order to turn again $[H;q']$ in canonical form.

Generation and Reception of administrative request. In Algorithm 4, the policy copy maintains a version counter that is incremented by the request generated by the administrator and performed on this copy. This request is next broadcast to other users for enforcing the new policy. When the received request r is causally ready (*i.e.* $r.v = version + 1$ and if r is a validation of a cooperative request q then this one has been already executed on this site), it is extracted from Q. If $r.o$ is *AddAuth* or *DelAuth*: (i) it is performed on the the policy copy; and, (ii) it undoes the tentative cooperative request that are no longer granted by the new policy. However, if r is a validation of cooperative request q then it sets q to valid.

```
1: RECEIVE_COOP_REQUEST(q):
2: if q is causally ready then
3:     F ← F − q
4:     if (CHECK_REMOTE(q,L)) then
5:         if i = adm then
6:             q.f ← valid
7:             r ← GENERATE_ADMIN_REQUEST(Validate(q))
8:         end if
9:     else
10:        q.f ← invalid
11:    end if
12:    q' ← COMPUTEFF (q,H)
13:    D ← Do(q',D)
14:    CANONIZE(q',H)
15: end if
```

Algorithm 3: Cooperative request reception by a site s

Asymptotic Time Complexities. Let H_{du} be all deletion/update requests H. In the worst case, when cooperative request q is an insertion and it has no dependency inside H (see [4]): (i) functions COMPUTEFF(q,H) and COMPUTEBF(q,H) have the same complexity, $O(|H|)$, and; (ii) the complexity of function CANONIZE(q,H) is $O(|H_{du}|)$. Hence, the complexity of GENERATE_COOP_REQUEST is $O(|H| + |H_{du}| + |Pr_v|) = O(2 * |H| + |Pr_v|)$ (with Pr_v is the list of autho-

```
1: GENERATE_ADMIN_REQUEST:
2: version ← version + 1
3: apply modification to the policy
4: r ← (admin, version, o)
5: broadcast r;

6: RECEIVE_ADMIN_REQUEST(r):
7: if r is causally ready then
8:     Q ← Q − r
9:     if (r.o is an AddAuth or DelAuth) then
10:        apply modification to policy
11:        if r is restrictive then
12:            H ← UNDO(q, H) for all tentative request q concerned by the request r
13:        end if
14:    else
15:        j ← GetIndex(r.q) {to determine the index of the cooperative request to validate it}
16:        H[j].f ← valid
17:    end if
18:    version ← version + 1
19: end if
```

Algorithm 4: Generation and reception of administrative request

rizations at version *v*), and the complexity of RECEIVE_COOP_REQUEST is $O(|L|+|H|+|H_{du}|) = O(|L|+2*|H|)$ (where L is the administrative log). Consequently, our concurrency control algorithm is not expensive and scale well as all functions have a linear behaviour.

However, to enforce the new authorization policy we have used the function UNDO(q,H). The complexity of this function is $O(|H|^2)$ when all H's requests are tentative and they should be undone by request r. Practically, UNDO is not expensive if we assume that the transmission time of requests is very short. In this case, the most of tentative requests will be validated by the administrator and there will be fewer requests to undo between two version of the policy object.

5.3 Illustrative Example

To highlight the feature of our concurrency control algorithm, we present a slightly complicated scenario in Figure 5, where the solid (dotted) arrows describe the integration order (validation of tentative requests). We have an administrator *adm* and two users s_1 and s_2 starting the collaboration with the initial state $D_0 =$ ''abc" and the initial policy version ($v_i^0 = v_0$) characterized by the policy $P_i^0 =< (All, Doc, \{iR, dR, rR, uR\}, +) >$ (for $i = adm, 1, 2$). The notations *All* and *Doc* designate the set of all users and the whole document respectively. Initially, the cooperative and administrative logs of each site are empty ($H_i^0 = L_i^0 = []$ for $i = adm, 1, 2$).

They generate three concurrent cooperative requests respectively: $q_0.o = Ins(2,y)$, $q_1.o = Del(2,b)$ and $q_2.o = Ins(3,x)$. After integrating q_0, q_1 and q_2, s_1 generates $q_3.o = Del(1,a)$. As for s_2, it generates $q_4.o = Del(1,a)$ after the integration of q_1 and q_2. Finally *adm* generates the administrative request $r.o = AddAuth(1,(s_1, Doc, dR, -))$. At the end of the collaboration, the three sites will converge to the final state "*ayc*".

We describe the integration of our requests in three steps:

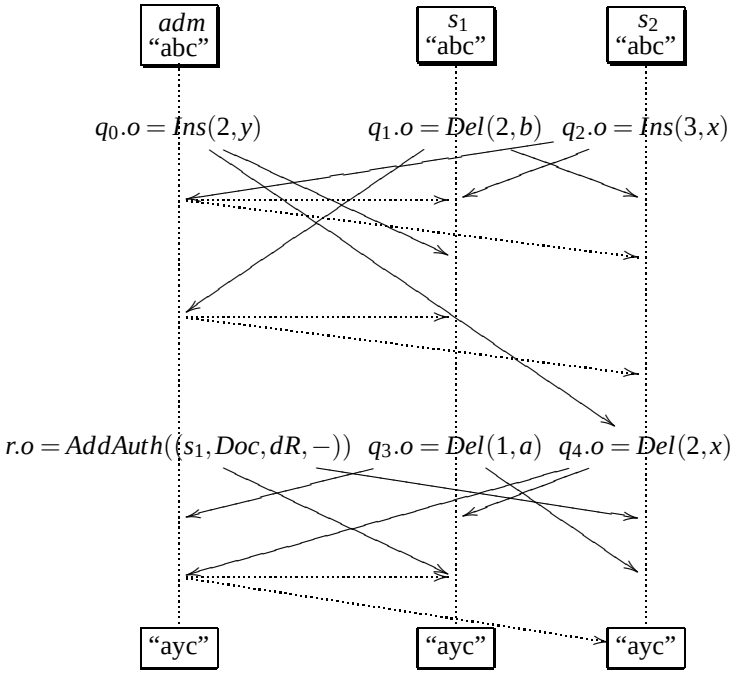


Figure 5: Collaboration scenario between an administrator and two sites.

Step 1. At *adm*, the execution of q_0 produces $D_0^1 =$ "*aybc*" and $H_0^1 = [q_0]$. When q_2 and q_1 arrive, they are transformed by COMPUTEFF(). This results in $D_0^3 = ayxc$ and $H_0^3 = [q_0; q_2'; q_1']$ with $q_2'.o = Ins(4,x)$ and $q_1'.o = Del(4,b)$. These requests are validated and sent to s_1 and s_2.

At s_1, the execution of q_1 gives $D_1^1 =$ "*ac*" and $H_1^1 = [q_1]$. Once received and granted by the local policy, q_2 and q_0 are transformed and the obtained log is twice modified by CANONIZE() as insertions must appear before deletions. We get $D_1^3 =$ "*ayxc*" and $H_1^3 = [q_2; q_0; q_1']$ with $q_1'.o = Del(3,b)$. Executing q_2 and q_1 at s_2 produces $D_2^2 =$ "*axc*" and $H_2^2 = [q_2; q_1]$.

The sites *adm*, s_1 and s_2 generate r, q_3 and q_4 respectively. They are propagated as follows.

Step 2. At *adm* site, r is restrictive and it produces $P_0^1 =< (s_1, Doc, dR, -), (All, Doc, \{iR, dR, rR, uR\}, +) >$, $L_0^1 = [r]$ and $v_0^1 = v_0 + 1$. Indeed, it revokes the deletion right to s_1.

At s_1, the execution of q_3 after H_1^3 results in $D_1^4 =$ "*yxc*". To broadcast q_3 with a minimal generation context, function COMPUTEBF() is called to detect causal dependency inside H_1^3. The obtained log is $H_1^4 = [q_2; q_0; q_1', q_3]$.

At s_2, q_4 is executed after H_2^2 and produces $D_2^3 =$ "*ac*" $H_2^3 = [q_2; q_1; q_4]$. Using COMPUTEBF() enables to detect that q_4 depends on q_2, as q_4 removes the character inserted by q_2. When q_0 arrives, its integration produces $D_2^4 =$ "*ayc*" and $H_2^4 = [q_2; q_0; q_1''; q_4'']$ (with $q_1''.o = Del(3,b)$ and $q_4''.o = Del(3,x)$). This log is the result of CANONIZE().

Step 3. At *adm*, when q_3 is checked against L_0^1 it is rejected but is stored in invalid form q_3^* which has no effect on the local document state. The resulting log is $H_0^5 = [q_0; q_2'; q_1', q_3^*]$. When q_4 arrives, it is only transformed against q_1' and q_3^* as it depends on q_2'. This results in $D_0^6 =$ "*ayc*" and $H_0^6 = [q_0; q_2'; q_1', q_3^*, q_4']$ with $q_4'.o = Del(3,x)$.

At s_1, the integration of q_4 produces $D_1^5 =$ "*yc*" and $H_1^5 = [q_2; q_0; q_1'; q_3; q_4]$. Integrating r results in $L_1^1 = [r]$ and $v_1^1 = v_0 + 1$. Enforcing the new policy requires to undo q_3 as it is a tentative (not validated yet) request. The inverse of q_3, noted $\overline{q_3}$, is firstly generated with $\overline{q_3}.o = Ins(1,a)$. Next, $\overline{q_3}$ is transformed against q_4 giving $\overline{q_3}'$ of which the execution results in $D_1^6 =$ "*ayc*$''$. Finally the log is modified to $H_1^6 = [q_2; q_0; q_1'; q_3; \overline{q_3}; q_4']$ where q_4' is the form of q_4 as if q_3 hasn't been executed.

At s_2, the reception of r results in $L_1^2 = [r]$ and $v_1^2 = v_0 + 1$. Request q_3 is invalidated ($q_3.f = invalid$) and stored in log without being executed. This results in $H_2^5 = [q_2; q_0; q_1'; q_4; q_3^*]$.

6 Implementation and Evaluation

A prototype of DCE based on our optimistic MAC has been implemented in Java. It supports the collaborative editing of HTML pages and it is deployed on P2P JXTA platform (see Figure 6). In our prototype, a user can create a HTML page from scratch by opening a new collaboration group. Thus, he is the administrator of this group. Others users may join the group to participate in HTML page editing, as they may leave this group at any time. The administrator can dynamically add and remove different authorizations for accessing to the shared document according the contribution and the competence of users participating in the group. Using JXTA platform, users exchange their operations in real-time in order to support WYSIWIS (What You See Is What I See) principle.

Experiments are necessary to understand what the asymptotic complexities mean when interactive constraints are present in the system. For our evaluation performance, we consider the following times: (i) t_1 is the execution time of *Generate_Coop_request*(); (ii) t_2 is the execution time of *Receive_Coop_request*(). We assume that the transmission time between sites is negligible. In general, it is established that the OT-based DCE must provide $t_1 + t_2 < 100ms$ [6]. Both algorithms 2 and 3 call function CANONIZE, their performances are mostly determined by the percentage of insertion requests inside the log. The management of the policy may affect the performance of the system since, in Algorithms 2 and 3, we have to explore either the policy or the the administrative log which are edited by the administrator. In our experiments we suppose that the policy is not optimized (*i.e.* it contains authorization redundancies).

Figure 7 shows three experiments[2] with different percentages of insertions inside log H. These measurements reflects the times t_1, t_2 and their sum. The execution time falls within 100ms for all $|H| \leq 5000$ if H contains 0% INS, $|H| \leq 9000$ if H contains 100% INS which is not achieved in SDT and ABT algorithms [6].

[2]The experiments have been performed under Ubunto Linux kernel 2.6.24-19 with an Intel Pentium-4 2.60 GHz CPU and 768 Mo RAM.

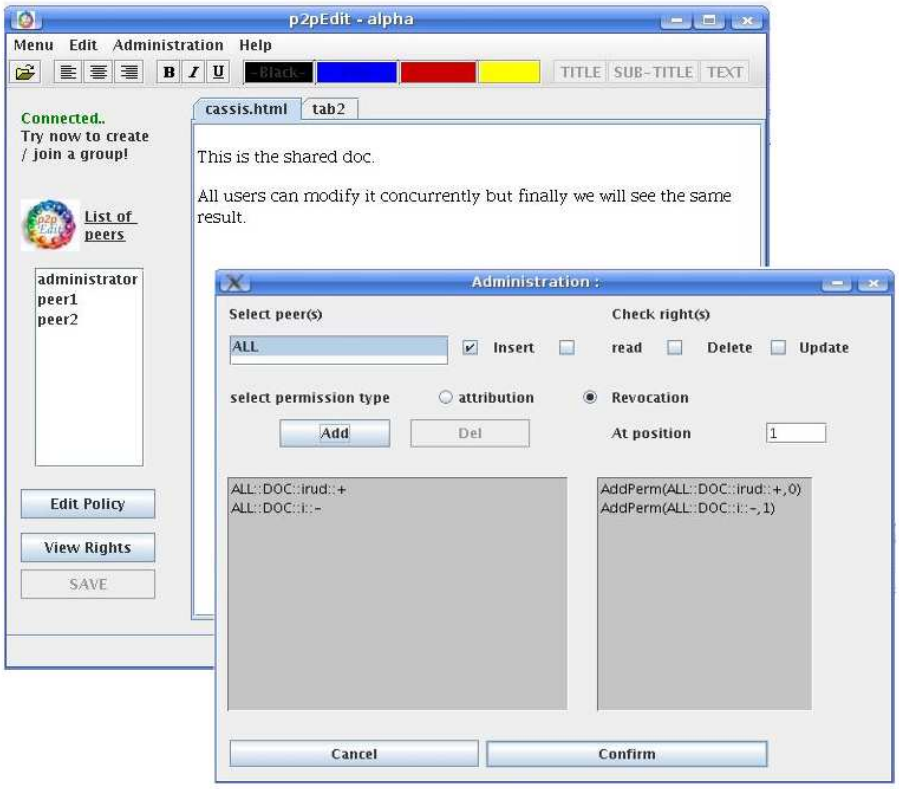


Figure 6: p2pEdit tool

7 Conclusion

In this paper, we have proposed a new framework for controlling access in collaborative editing work. It is based on MAC and optimistic replication of the shared document and its authorization policy. We have shown how naive coordination between updates of both copies may create security holes. Finally, we have provided some performance evaluations to show the applicability of our MAC model distributed collaborative editing.

In future work, we intend to investigate the impact of our work when using delegation of administrative requests between the group users. As the length of local (administrative and cooperative) logs increases rapidly during collaboration sessions, we plan to address the garbage collection problem.

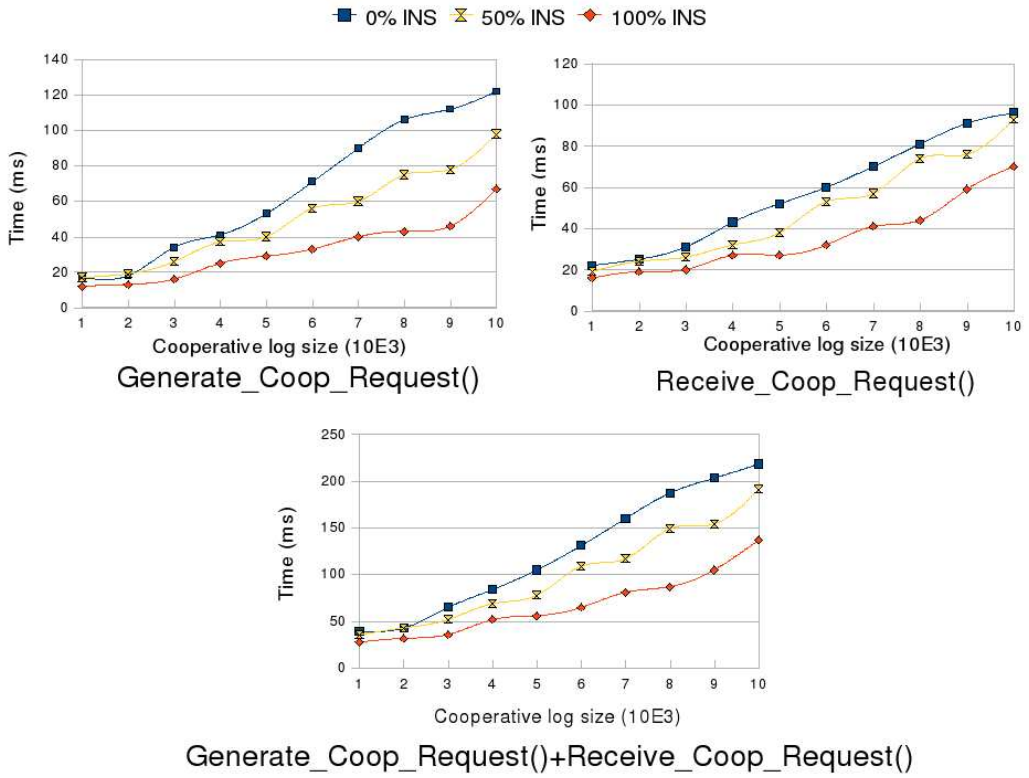


Figure 7: Time processing of Insert Requests.

References

[1] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. A decentralized temporal autoritzation model. In *SEC*, pages 271–280, 1996.

[2] A. Bullock and S. Benford. An access control framework for multi-user collaborative environments. In *GROUP '99*, pages 140–149, New York, NY, USA, 1999. ACM.

[3] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[4] A. Imine. Flexible Concurrency Control for Real-time Collaborative Editors. In *28th International Conference on Distributed Computing Systems Workshops, ICDCSW*, pages 423–428, Beijing, China, June 2008. IEEE Computer Society.

[5] T. Jaeger and A. Prakash. Requirements of role-based access control for collaborative systems. In *RBAC '95*, page 16, New York, NY, USA, 1996. ACM.

[6] D. Li and R. Li. An operational transformation algorithm and performance evaluation. *Computer Supported Cooperative Work*, 17(5-6):469–508, 2008.

[7] B. Lushman and G. V. Cormack. Proof of correctness of ressel's adopted algorithm. *Information Processing Letters*, 86(3):303–310, 2003.

[8] D. Povey. Optimistic security: a new access control paradigm. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 40–45. ACM, 2000.

[9] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhauser. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *ACM CSCW'96*, pages 288–297, Boston, USA, November 1996.

[10] P. Samarati, P. Ammann, and S. Jajodia. Maintaining replicated authorizations in distributed database systems. *Data Knowl. Eng.*, 18(1):55–84, 1996.

[11] P. Samarati and S. D. C. di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD*, pages 137–196, 2000.

[12] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[13] H. Shen and P. Dewan. Access control for collaborative environments. In *CSCW '92*, pages 51–58, New York, NY, USA, 1992. ACM.

[14] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *ACM CSCW'98*, pages 59–68, 1998.

[15] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality-preservation and Intention-preservation in real-time Cooperative Editing Systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[16] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, 2006.

[17] W. Tolone, G.-J. Ahn, T. Pai, and S.-P. Hong. Access control in collaborative systems. *ACM Comput. Surv.*, 37(1):29–41, 2005.

[18] T. Xin and I. Ray. A lattice-based approach for updating access control policies in real-time. *Inf. Syst.*, 32(5):755–772, 2007.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399